

Gaining Insight into Parallel Program Performance using HPCToolkit

John Mellor-Crummey
Department of Computer Science
Rice University

<http://hpctoolkit.org>



Challenges for Computational Scientists

- **Rapidly evolving platforms and applications**
 - **architecture**
 - rapidly changing multicore microprocessor designs
 - increasing architectural diversity
 - CPU, GPU, APU, manycore (e.g., Xeon Phi)
 - increasing scale of parallel systems
 - **applications**
 - augment computational capabilities
- **Computational scientist needs**
 - adapt to changes in emerging architectures
 - adding threading and/or offloading to accelerators
 - improve scalability within and across nodes
 - assess weaknesses in algorithms and their implementations

Performance tools can play an important role as a guide

Performance Analysis Challenges

- **Complex node architectures are hard to use efficiently**
 - multi-level parallelism: multiple cores, ILP, SIMD, accelerators
 - multi-level memory hierarchy
 - result: gap between typical and peak performance is huge
- **Complex applications present challenges**
 - measurement and analysis
 - understanding behaviors and tuning performance
- **Supercomputer platforms compound the complexity**
 - unique hardware & microkernel-based operating systems
 - multifaceted performance concerns
 - computation
 - data movement
 - communication
 - I/O

What Users Want

- **Easy-to-use multi-platform, programming model independent tools**
- **Accurate measurement of complex parallel codes**
 - large, multi-lingual programs
 - (heterogeneous) parallelism within and across nodes
 - optimized code: loop optimization, templates, inlining
 - binary-only libraries, sometimes partially stripped
 - complex execution environments
 - dynamic binaries on clusters; static binaries on supercomputers
 - batch jobs
- **Effective performance analysis**
 - insightful analysis that pinpoints and explains problems
 - correlate measurements with code for actionable results
 - support analysis at the desired level
 - intuitive enough for application scientists and engineers
 - detailed enough for library developers and compiler writers
- **Scalable to petascale and beyond**

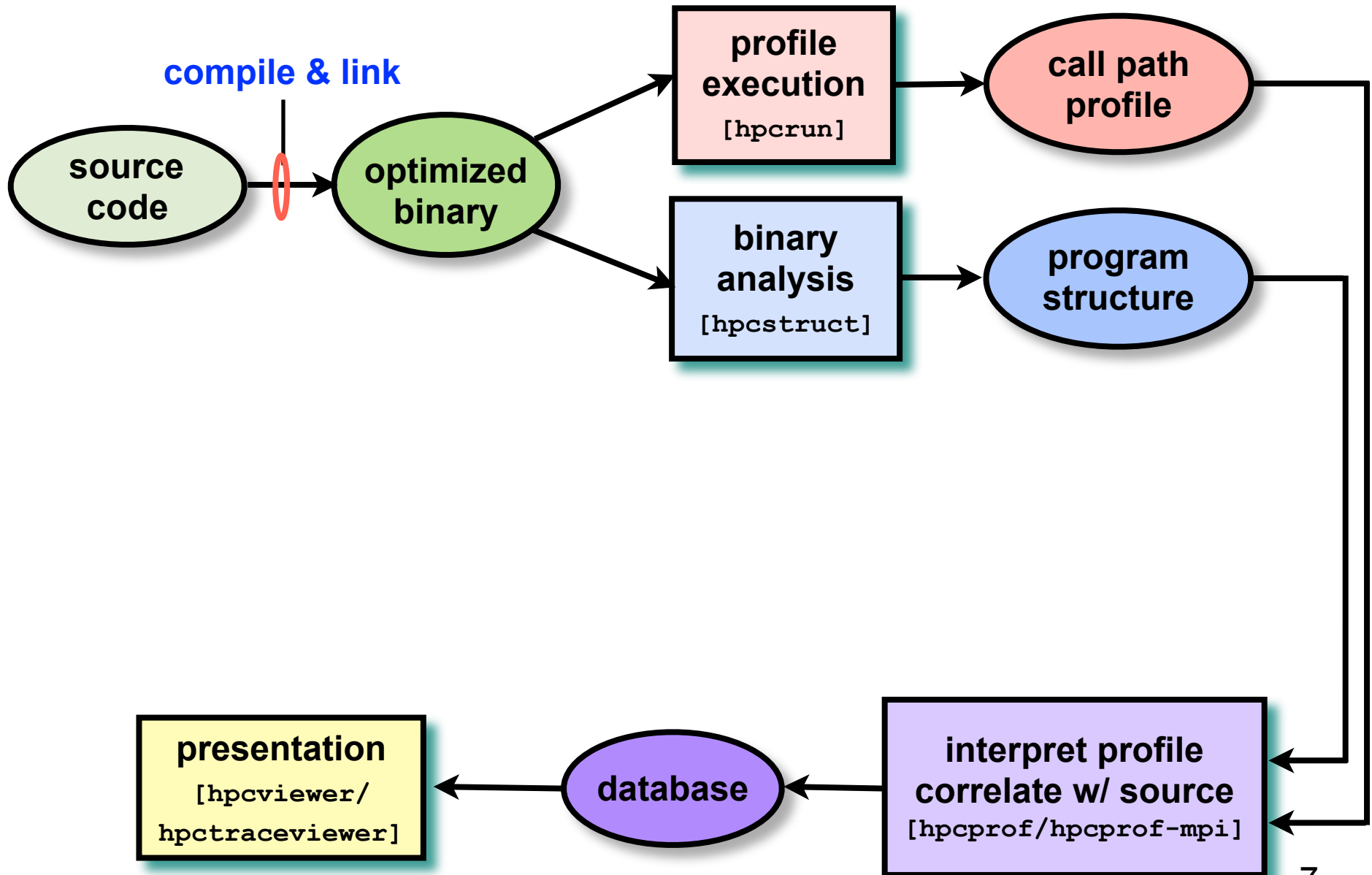
Rice University's HPCToolkit

- Employs binary-level measurement and analysis
 - observe executions of **optimized code**
 - support **multi-lingual codes** with external binary-only libraries
- Uses sampling-based measurement (avoid instrumentation)
 - **controllable overhead**
 - **minimize** systematic error and avoid blind spots
 - enable data collection for **large-scale parallelism**
- Collects and correlates multiple derived performance metrics
 - diagnosis typically requires more than one species of metric
- Associates metrics with both static and dynamic context
 - **loop nests**, **procedures**, **inlined code**, **calling context**
- Supports top-down performance analysis
 - natural approach that minimizes burden on developers

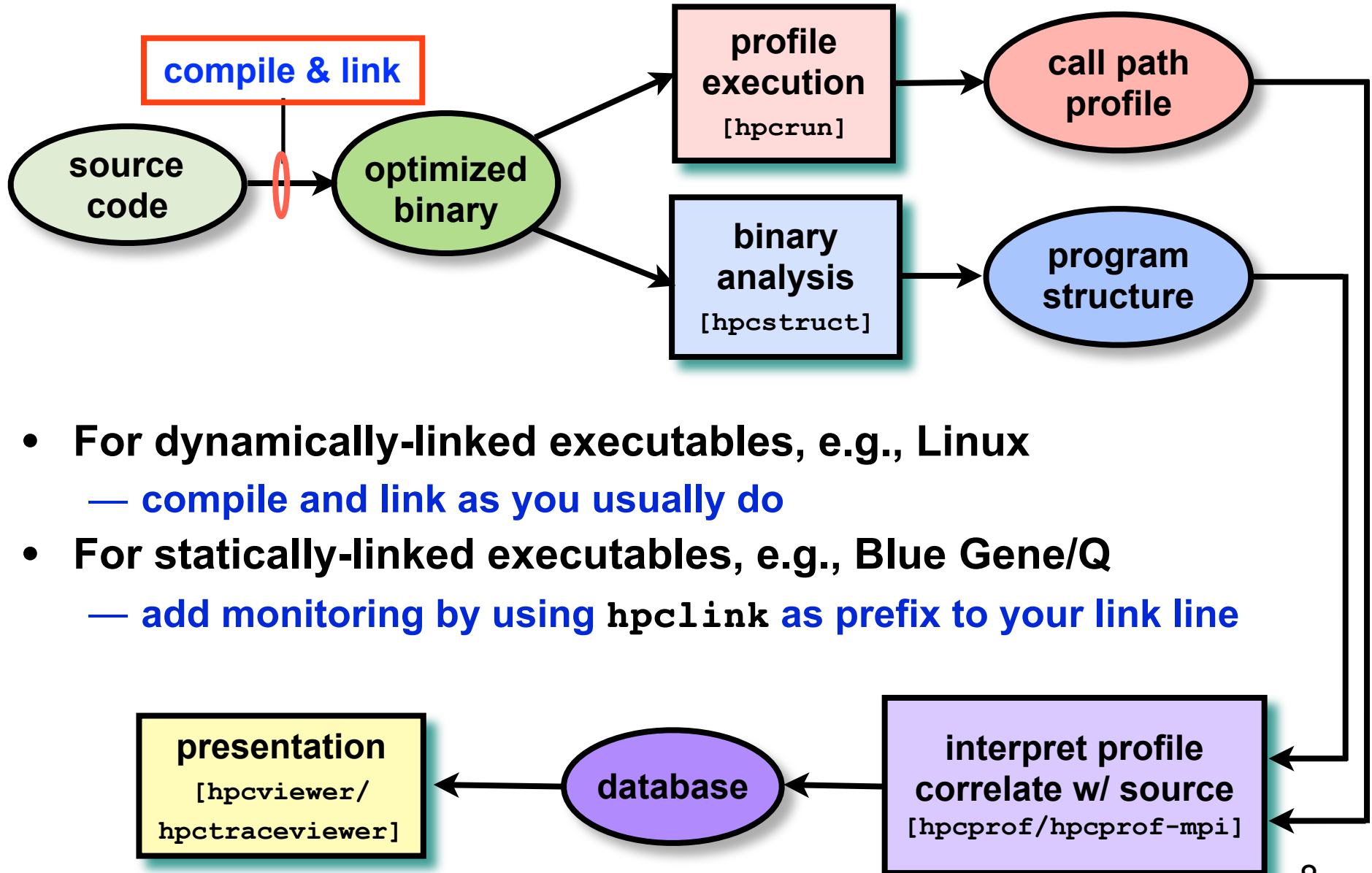
Outline

- **Overview of Rice's HPCToolkit**
- **Pinpointing scalability bottlenecks**
 - scalability bottlenecks on large-scale parallel systems
 - scaling on multicore processors
- **Understanding temporal behavior**
- **Assessing process variability**
- **Understanding threading, GPU, and memory hierarchy**
 - blame shifting
 - attributing memory hierarchy costs to data
- **Summary and challenges ahead**

HPCToolkit Workflow

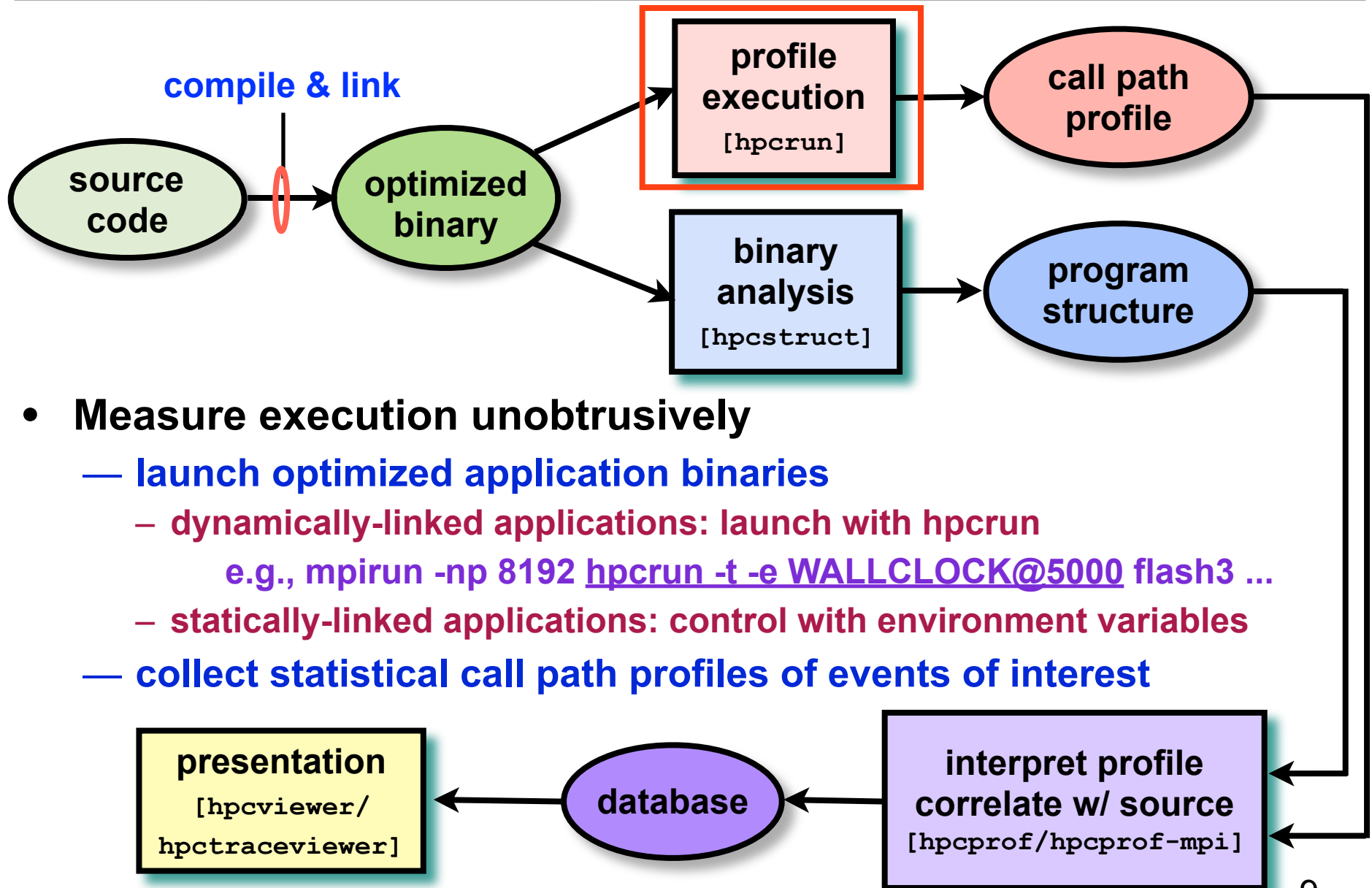


HPCToolkit Workflow



- For dynamically-linked executables, e.g., Linux
 - **compile and link as you usually do**
- For statically-linked executables, e.g., Blue Gene/Q
 - **add monitoring by using `hpclink` as prefix to your link line**

HPCToolkit Workflow



- **Measure execution unobtrusively**
 - **launch optimized application binaries**
 - **dynamically-linked applications: launch with hpcrun**
e.g., `mpirun -np 8192 hpcrun -t -e WALLCLOCK@5000 flash3 ...`
 - **statically-linked applications: control with environment variables**
 - **collect statistical call path profiles of events of interest**

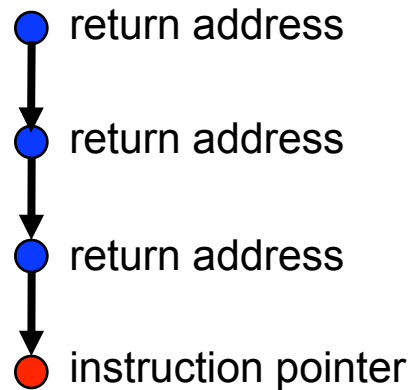
Call Path Profiling

Measure and attribute costs in context

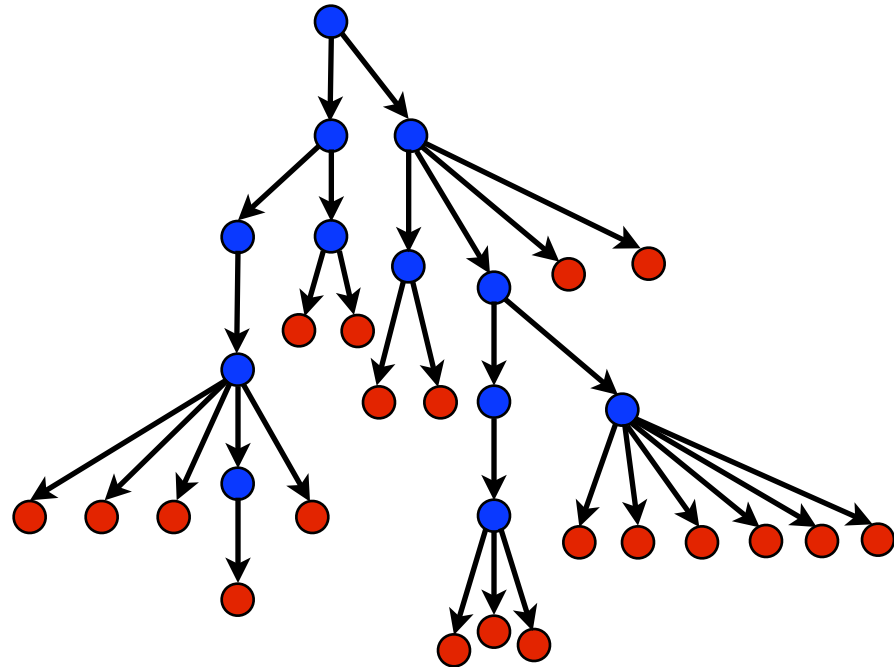
sample timer or hardware counter overflows

gather calling context using stack unwinding

Call path sample

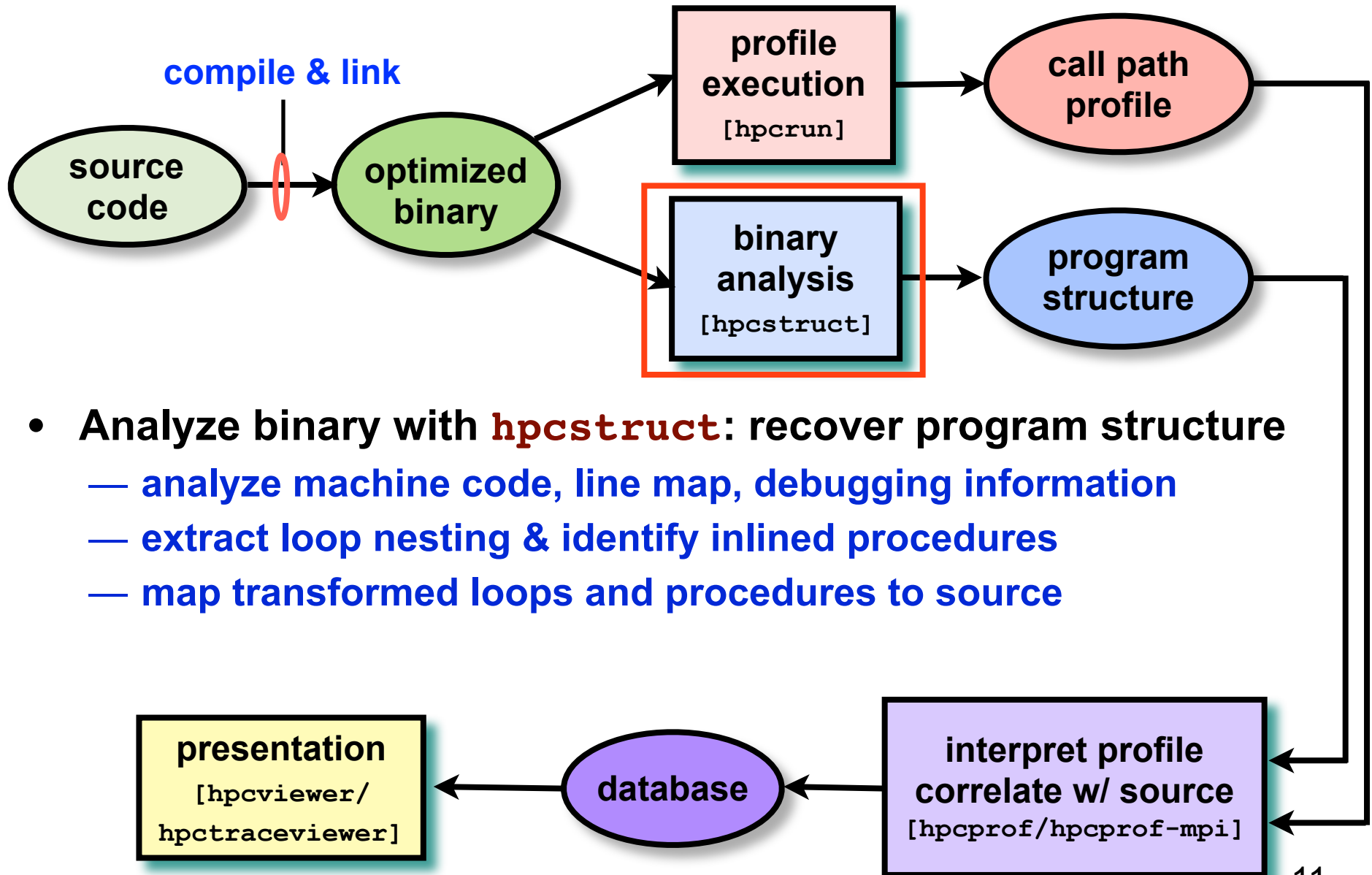


Calling context tree



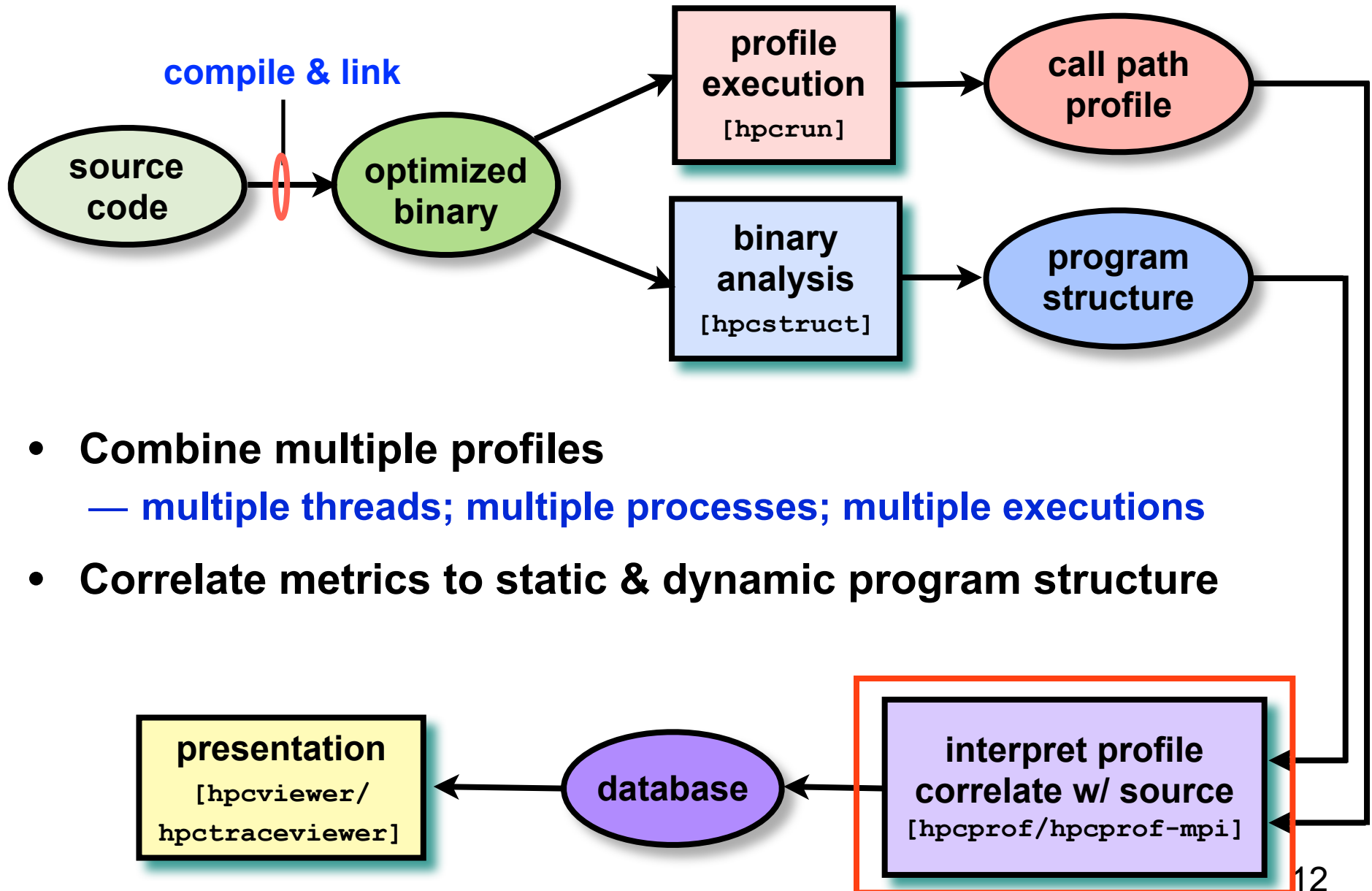
**Overhead proportional to sampling frequency...
...not call frequency**

HPCToolkit Workflow

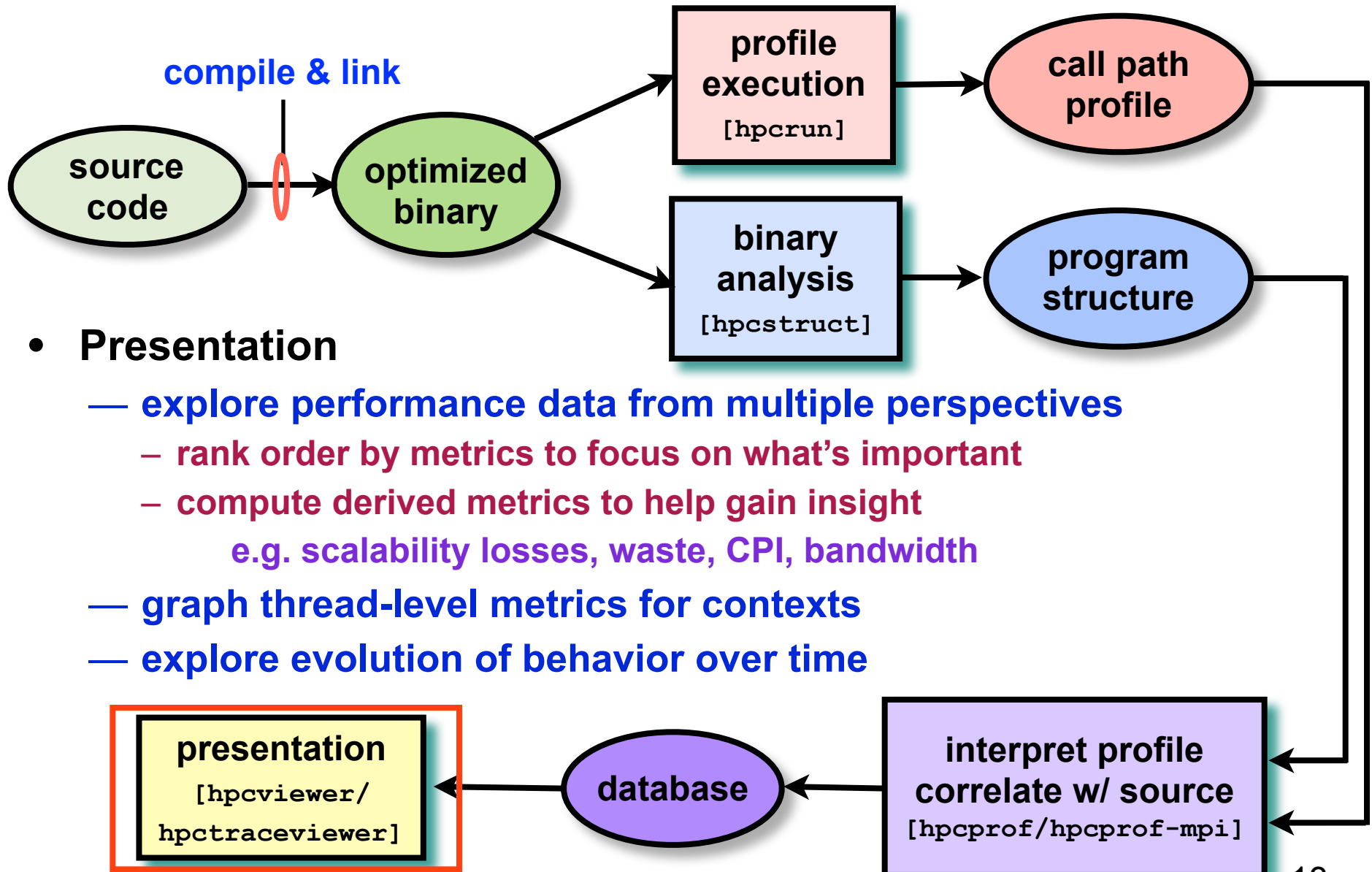


- Analyze binary with **hpcstruct**: recover program structure
 - analyze machine code, line map, debugging information
 - extract loop nesting & identify inlined procedures
 - map transformed loops and procedures to source

HPCToolkit Workflow



HPCToolkit Workflow



Analyzing Chombo@1024 cores with hpcviewer

The screenshot displays the hpcviewer application window titled "hpcviewer: amrGodunov3d.Linux.64.CC.ftn.OPTHIGH.MPI.ex". The interface is divided into several panes:

- source pane:** Shows the source code of "PatchGodunov.cpp" with line numbers 947 to 957. A red box highlights the "source pane" label.
- view control:** A toolbar with buttons for "Calling Context View", "Callers View", and "Flat View". A red box highlights the "view control" label.
- metric display:** A toolbar with icons for metrics. A red box highlights the "metric display" label.
- navigation pane:** A tree view showing the execution scope, including "Experiment Aggregate Metrics", "main", and various loops and function calls. A red box highlights the "navigation pane" label.
- metric pane:** A table showing performance metrics. A red box highlights the "metric pane" label.

costs for

- inlined procedures
- loops
- function calls in full context

Table Data:

Scope	WALLCLOCK (us):Sum (l)	WALLCLOCK (us):Mean (l)	WALLCLOCK (us):Min (l)
Experiment Aggregate Metrics	1.92e+11 100 %	1.80e+08	
main	1.92e+11 100 %	1.80e+08	
↳ 282: amrGodunov()	1.87e+11 97.4%	1.75e+08	
↳ loop at amrGodunov.cpp: 186	1.77e+11 92.1%	1.66e+08	
↳ loop at amrGodunov.cpp: 214	1.77e+11 92.1%	1.66e+08	
↳ 216: AMR::run(double, int)	1.77e+11 92.1%	1.66e+08	
↳ inlined from AMR.cpp: 604	1.77e+11 92.1%	1.66e+08	
↳ loop at AMR.cpp: 615	1.77e+11 92.1%	1.66e+08	
↳ loop at AMR.cpp: 622	1.77e+11 92.1%	1.66e+08	
↳ 654: AMR::timeStep(int, int, bool)	1.77e+11 92.1%	1.66e+08	
↳ inlined from AMR.cpp: 794	1.77e+11 92.1%	1.66e+08	
↳ loop at AMR.cpp: 943	1.77e+11 92.0%	1.66e+08	
↳ 953: AMR::timeStep(int, int, bool)	1.77e+11 92.0%	1.66e+08	
↳ inlined from AMR.cpp: 794	1.77e+11 92.0%	1.66e+08	
↳ 903: AMRLevelPolytropicGas::advance()	1.73e+11 90.3%	1.62e+08	
↳ 919: BoxLayout::size() const	5.37e+06 0.0%	5.04e+03	
↳ 911: AMRLevelPolytropicGas::computeDt()	2.04e+05 0.0%	1.91e+02	
AMR.cpp: 795	2.40e+04 0.0%	2.25e+01	
↳ 967: AMRLevelPolytropicGas::postTimeStep()	1.20e+04 0.0%	1.12e+01	
↳ 801: std::ostream& std::ostream::M_insert<long>(long)	1.20e+04 0.0%	1.12e+01	

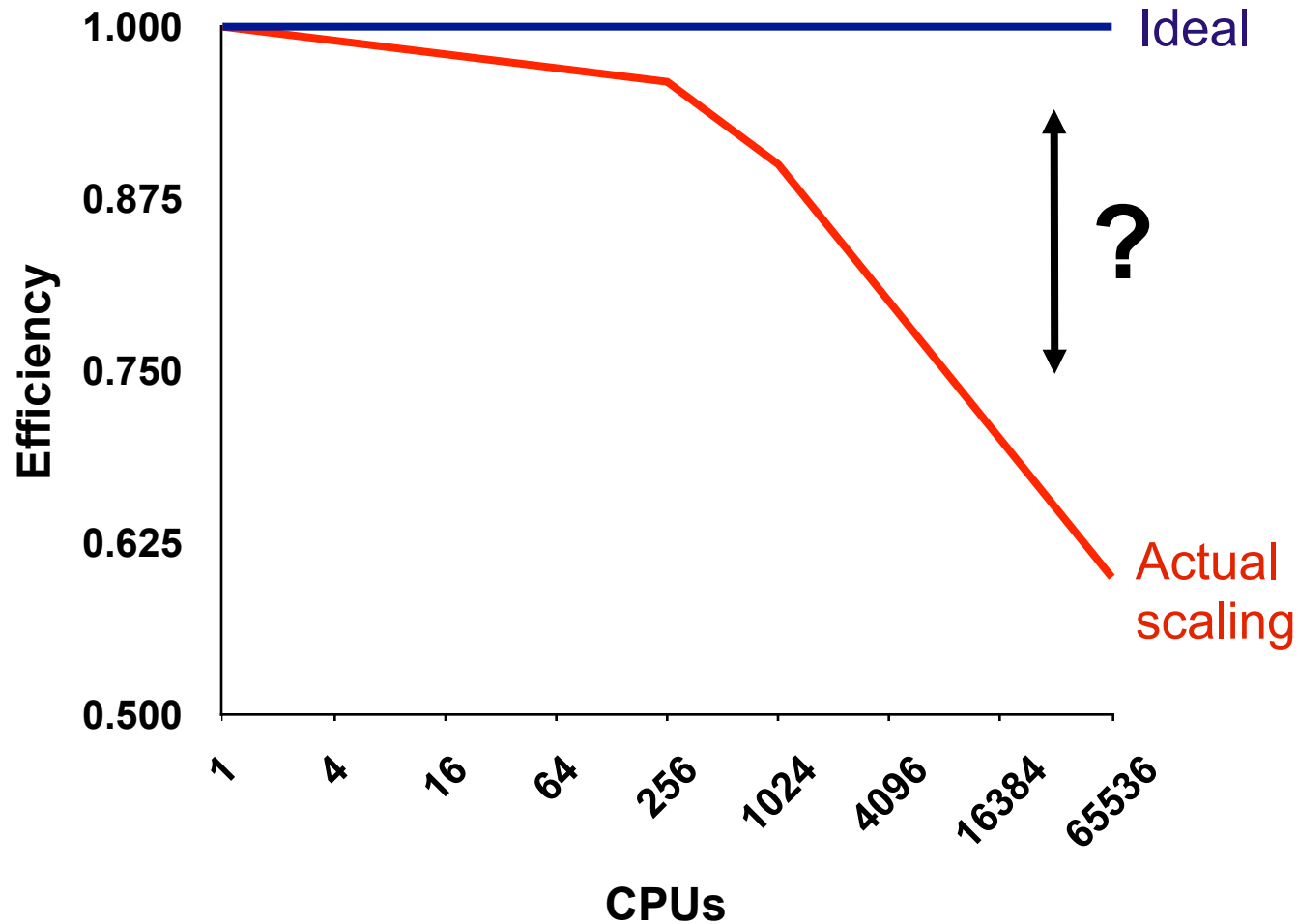
Principal Views

- **Calling context tree view - “top-down” (down the call chain)**
 - associate metrics with each dynamic calling context
 - high-level, hierarchical view of distribution of costs
 - example: quantify initialization, solve, post-processing
- **Caller’s view - “bottom-up” (up the call chain)**
 - apportion a procedure’s metrics to its dynamic calling contexts
 - understand costs of a procedure called in many places
 - example: see where PGAS put traffic is originating
- **Flat view - ignores the calling context of each sample point**
 - aggregate all metrics for a procedure, from any context
 - attribute costs to loop nests and lines within a procedure
 - example: assess the overall memory hierarchy performance within a critical procedure

Outline

- Overview of Rice's HPCToolkit
- Pinpointing scalability bottlenecks
 - scalability bottlenecks on large-scale parallel systems
 - scaling on multicore processors
- Understanding temporal behavior
- Assessing process variability
- Understanding threading, GPU, and memory hierarchy
 - blame shifting
 - attributing memory hierarchy costs to data
- Summary and challenges ahead

The Problem of Scaling



Note: higher is better

Wanted: Scalability Analysis

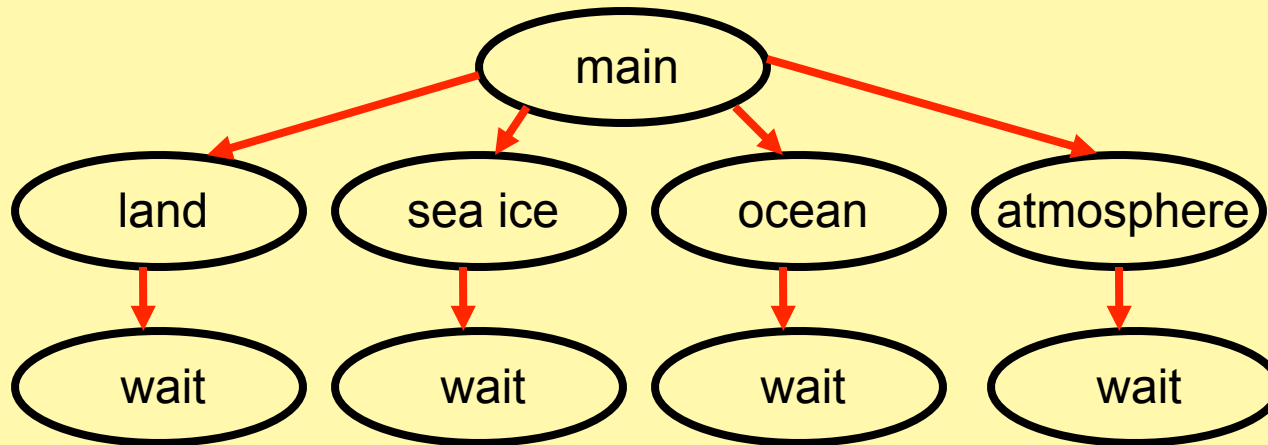
- **Isolate scalability bottlenecks**
- **Guide user to problems**
- **Quantify the magnitude of each problem**

Challenges for Pinpointing Scalability Bottlenecks

- **Parallel applications**

- modern software uses layers of libraries
- performance is often context dependent

Example climate code skeleton



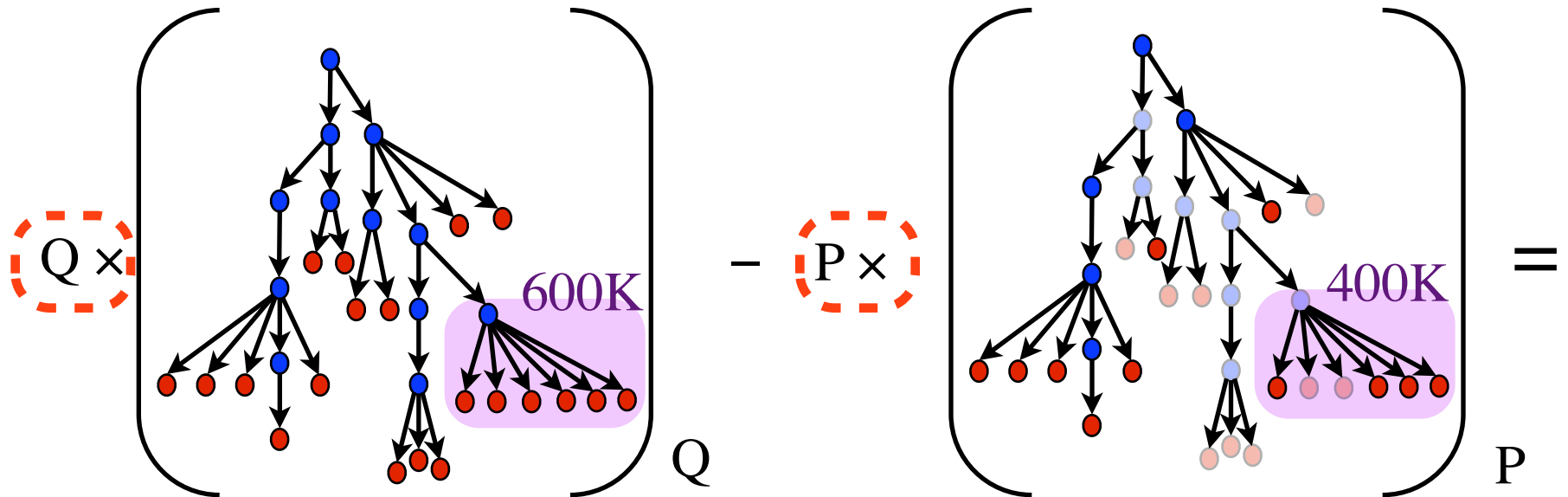
- **Monitoring**

- bottleneck nature: computation, data movement, synchronization?
- **2 pragmatic constraints**
 - acceptable data volume
 - low perturbation for use in production runs

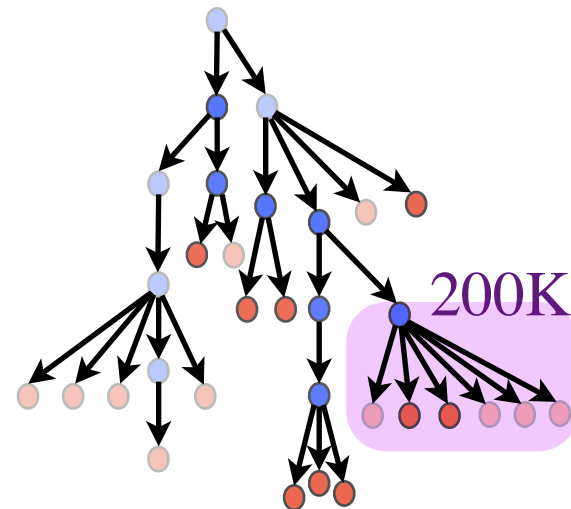
Performance Analysis with Expectations

- You have performance expectations for your parallel code
 - strong scaling: linear speedup
 - weak scaling: constant execution time
- Put your expectations to work
 - measure performance under different conditions
 - e.g. different levels of parallelism and/or different problem size
 - express your expectations as an equation
 - compute the deviation from expectations for each calling context
 - for both inclusive and exclusive costs
 - correlate the metrics with the source code
 - explore the annotated call tree interactively

Pinpointing and Quantifying Scalability Bottlenecks

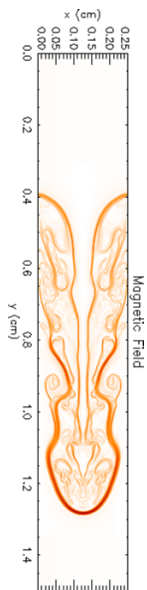


coefficients for analysis
of strong scaling

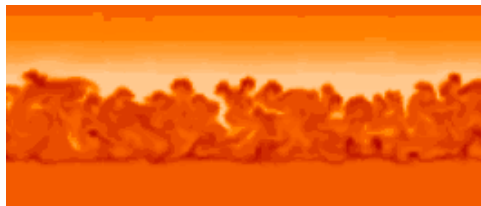


Scalability Analysis Demo: FLASH3

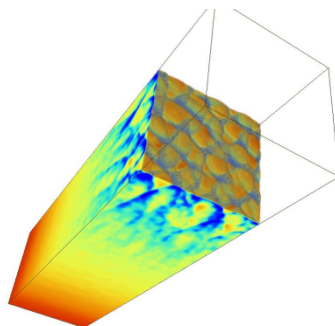
Code:	University of Chicago FLASH3
Simulation:	white dwarf detonation
Platform:	Blue Gene/P
Experiment:	8192 vs. 256 processors
Scaling type:	weak



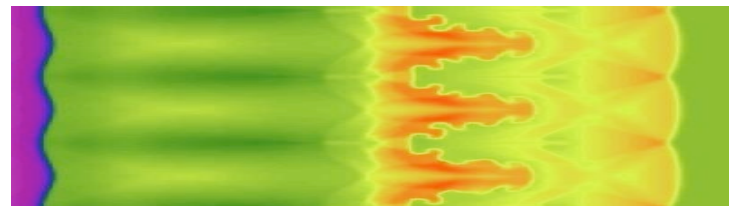
*Magnetic
Rayleigh-Taylor*



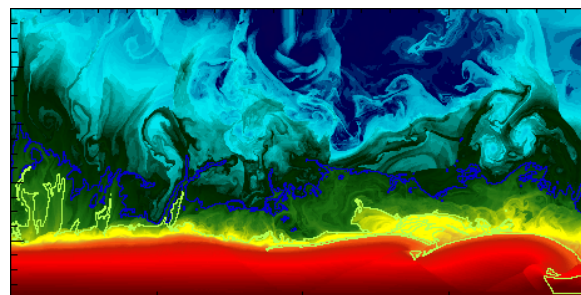
Nova outbursts on white dwarfs



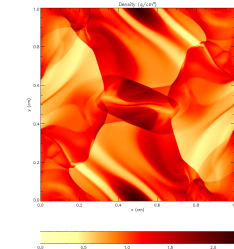
Cellular detonation



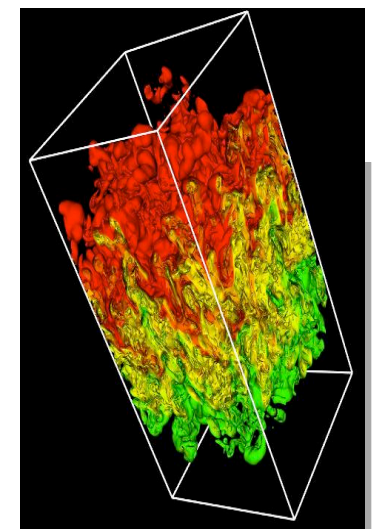
Laser-driven shock instabilities



Helium burning on neutron stars



*Orzag/Tang MHD
vortex*



Rayleigh-Taylor instability

Figures courtesy of FLASH Team, University of Chicago

Scalability Analysis of Flash3 (Demo)

hpcviewer: FLASH/white dwarf: IBM BG/P, weak 256->8192

Driver_initFlash.F90 local_tree_build.F90

```

206 !-----First pass only add lrefine = 1 blocks to tree(s)
207 !-----Second pass add the rest of the blocks.
208     Do ipass = 1,2
209
210         lnblocks_old = lnblocks
211         proc = mype
212 !-----Loop through all processors
213     Do iproc = 0, nprocs-1
214
215         If (iproc == 0) Then
216             off_proc = .False.
217         Else

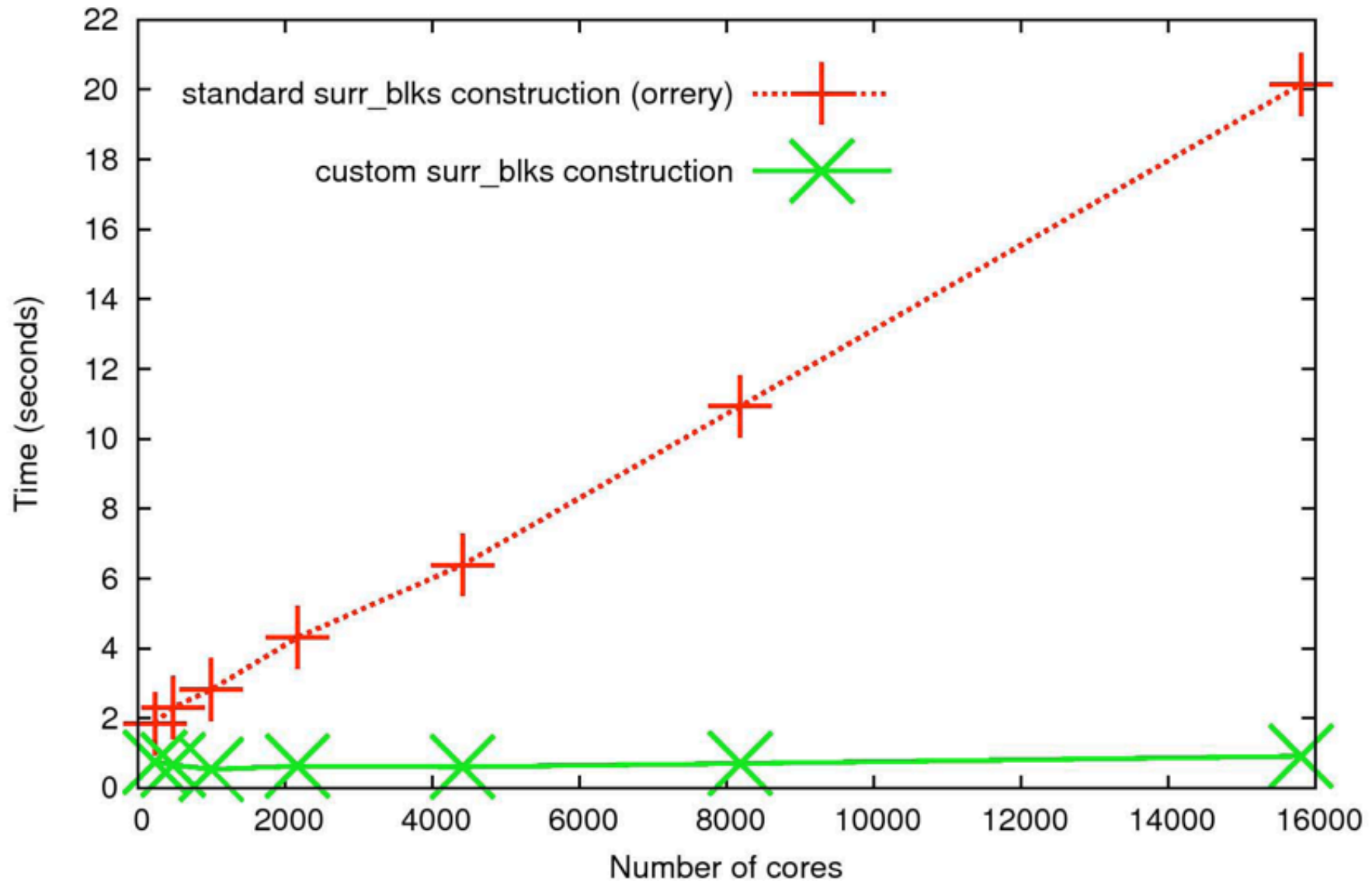
```

Calling Context View Callers View Flat View

Scope % scalability loss 256/WALLCLOCK (u)

Scope	% scalability loss	256/WALLCLOCK (u)
Experiment Aggregate Metrics	2.46e+01 100 %	5.07e+08
▼ flash	2.46e+01 100 %	5.07e+08
▶ driver_evolveflash	1.41e+01 57.5%	4.46e+08
▼ driver_initflash	1.04e+01 42.5%	6.02e+07
▼ grid_initdomain	8.58e+00 34.9%	3.45e+07
▼ gr_expanddomain	8.58e+00 34.9%	3.45e+07
▼ loop at gr_expandDomain.F90: 119	6.85e+00 27.9%	3.42e+07
▼ amr_refine_derefine	5.56e+00 22.6%	2.87e+06
▼ amr_morton_process	5.45e+00 22.2%	9.75e+05
▼ find_surrblks	5.18e+00 21.1%	8.40e+05
▼ local_tree_build	5.18e+00 21.1%	8.25e+05
▼ loop at local_tree_build.F90: 211	5.18e+00 21.1%	8.25e+05
▼ loop at local_tree_build.F90: 216	5.18e+00 21.1%	8.25e+05
▶ loop at local_tree_build.F90: 286	1.14e+00 4.6%	2.55e+05
▶ pmmpi_sendrecv_replace	5.47e-01 2.2%	5.00e+04

Improved Flash Scaling of AMR Setup



Graph courtesy of Anshu Dubey, U Chicago

Scaling on Multicore Processors

- **Compare performance**
 - single vs. multiple processes on a multicore system
- **Strategy**
 - differential performance analysis
 - subtract the calling context trees as before, unit coefficient for each

S3D: Multicore Losses at the Procedure Level

The screenshot shows the hpcviewer interface. The top pane displays the source code for the subroutine `rhsf`. The bottom pane shows a performance table with columns for Scope, 1-core (ms) (I), 1-core (ms) (E), 8-core(1) (ms) (I), 8-core(1) (ms) (E), and Multicore Loss. The `rhsf` row is highlighted, and its Multicore Loss is 13.0%.

```
1 subroutine rhsf( q, rhs )
2 !-----
3 ! Changes
4 ! Ramanan Sankaran - 01/04/05
5 ! 1. Diffusive fluxes are computed without having to convert units.
6 ! Ignore older comments about conversion to CGS units.
7 ! This saves a lot of flops.
8 ! 2. Mixavg and Lewis transport modules have been made interchangeable
9 ! by adding dummy arguments in both.
10 !-----
11 !           Author: James Sutherland
12 !           Date:   April, 2002
13 !-----
14 ! This routine calculates the time rate of change for the
15 ! momentum, continuity, energy, and species equations.
16 !
```

Scope	1-core (ms) (I)	1-core (ms) (E)	8-core(1) (ms) (I)	8-core(1) (ms) (E)	Multicore Loss
Experiment Aggregate Metrics	1.11e08 100 %	1.11e08 100 %	1.88e08 100 %	1.88e08 100 %	7.64e07 100 %
rhsf	1.07e08 96.5%	6.60e06 5.9%	1.77e08 94.1%	1.65e07 8.8%	9.92e06 13.0%
diffflux_proc_looptool	2.86e06 2.6%	2.86e06 2.6%	8.12e06 4.3%	8.12e06 4.3%	5.27e06 6.9%
integrate_erk_jstage_lt	1.09e08 98.1%	1.25e06 1.1%	1.84e08 97.9%	5.94e06 3.2%	4.70e06 6.1%
GET_MASS_FRAC.in.VARIABLES_M	1.49e06 1.3%	1.49e06 1.3%	6.08e06 3.2%	6.08e06 3.2%	4.59e06 6.0%
ratx	1.01e07 9.1%	1.00e07 9.0%	4.41e07 23.5%	1.40e07 7.4%	3.95e06 5.2%
qssa	3.52e06 3.2%	3.52e06 3.2%	5.71e06 3.0%	5.71e06 3.0%	2.18e06 2.9%
ratt	3.26e07 29.2%	1.48e07 13.3%	4.38e07 23.3%	1.66e07 8.8%	1.76e06 2.3%
CALC_INV_AVG_MOL_WT.in.THER	9.70e05 0.9%	9.70e05 0.9%	2.68e06 1.4%	2.68e06 1.4%	1.70e06 2.2%
computeheatflux_looptool	1.46e06 1.3%	1.46e06 1.3%	2.88e06 1.5%	2.88e06 1.5%	1.41e06 1.8%
rdwdot	3.09e06 2.8%	3.09e06 2.8%	4.33e06 2.3%	4.33e06 2.3%	1.24e06 1.6%

Execution time increases 1.65x in subroutine `rhsf`

subroutine `rhsf` accounts for 13.0% of the multicore scaling loss in the execution

S3D: Multicore Losses at the Loop Level

The screenshot shows the hpcviewer interface. The top pane displays Fortran code from the file `diffflux_gen_uj.f`. The bottom pane shows the 'Flat View' of the performance profile, which is a table with columns for Scope, 1-core (ms) (I), 1-core (ms) (E), 8-core(1) (ms) (I), 8-core(1) (ms) (E), and Multicore Loss. A red arrow points from the first row of the table to the corresponding code block in the top pane.

Execution time increases 2.8x in the loop that scales worst

loop contributes 6.9% of the scaling loss for the whole execution

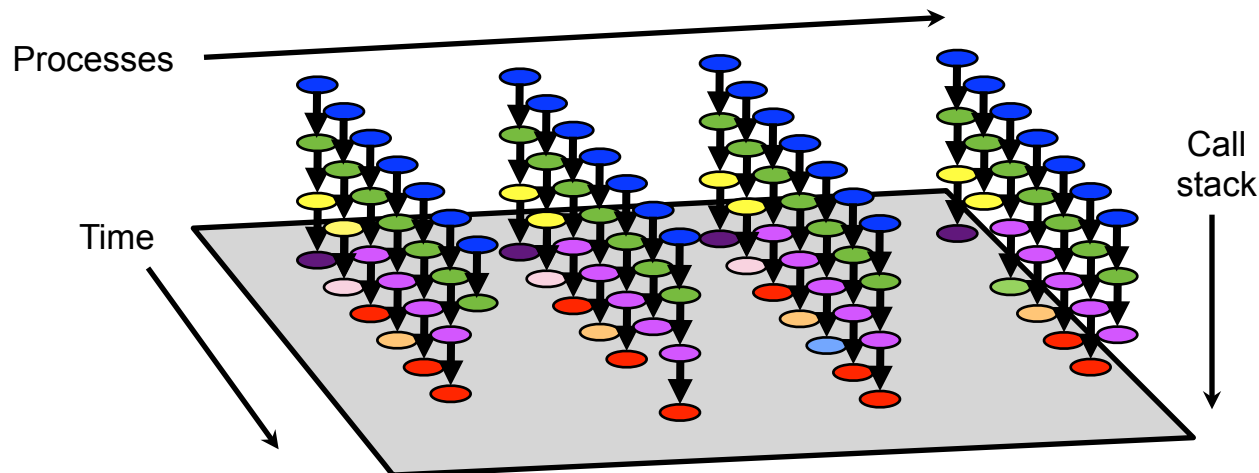
Scope	1-core (ms) (I)	1-core (ms) (E)	8-core(1) (ms) (I)	8-core(1) (ms) (E)	Multicore Loss
loop at diffflux_gen_uj.f: 197-223	2.86e06 2.6%	2.86e06 2.6%	8.12e06 4.3%	8.12e06 4.3%	5.27e06 6.9%
loop at integrate_erk_jstage_lt_ge	1.09e08 98.1%	1.25e06 1.1%	1.84e08 97.9%	5.94e06 3.2%	4.70e06 6.1%
loop at variables_m.f90: 88-99	1.49e06 1.3%	1.49e06 1.3%	6.08e06 3.2%	6.08e06 3.2%	4.60e06 6.0%
loop at rhsf.f90: 516-536	2.70e06 2.4%	1.31e06 1.2%	6.49e06 3.5%	3.72e06 2.0%	2.41e06 3.1%
loop at rhsf.f90: 538-544	3.35e06 3.0%	1.45e06 1.3%	7.06e06 3.8%	3.82e06 2.0%	2.36e06 3.1%
loop at rhsf.f90: 546-552	2.56e06 2.3%	1.47e06 1.3%	5.86e06 3.1%	3.42e06 1.8%	1.96e06 2.6%
loop at thermchem_m.f90: 127-1	8.00e05 0.7%	8.00e05 0.7%	2.28e06 1.2%	2.28e06 1.2%	1.48e06 1.9%
loop at heatflux_lt_gen.f: 5-132	1.46e06 1.3%	1.46e06 1.3%	2.88e06 1.5%	2.88e06 1.5%	1.41e06 1.8%
loop at rhsf.f90: 576	6.65e05 0.6%	6.65e05 0.6%	1.87e06 1.0%	1.87e06 1.0%	1.20e06 1.6%
loop at getrates.f: 504-505	8.00e06 7.2%	8.00e06 7.2%	8.74e06 4.7%	8.74e06 4.7%	7.35e05 1.0%
loop at derivative_x.f90: 213-690	1.78e06 1.6%	1.78e06 1.6%	2.47e06 1.3%	2.47e06 1.3%	6.95e05 0.9%

Outline

- Overview of Rice's HPCToolkit
- Pinpointing scalability bottlenecks
 - scalability bottlenecks on large-scale parallel systems
 - scaling on multicore processors
- Understanding temporal behavior
- Assessing process variability
- Understanding threading, GPU, and memory hierarchy
 - blame shifting
 - attributing memory hierarchy costs to data
- Summary and challenges ahead

Understanding Temporal Behavior

- Profiling compresses out the temporal dimension
 - temporal patterns, e.g. serialization, are invisible in profiles
- What can we do? Trace call path samples
 - sketch:
 - N times per second, take a call path sample of each thread
 - organize the samples for each thread along a time line
 - view how the execution evolves left to right
 - what do we view?
 - assign each procedure a color; view a depth slice of an execution

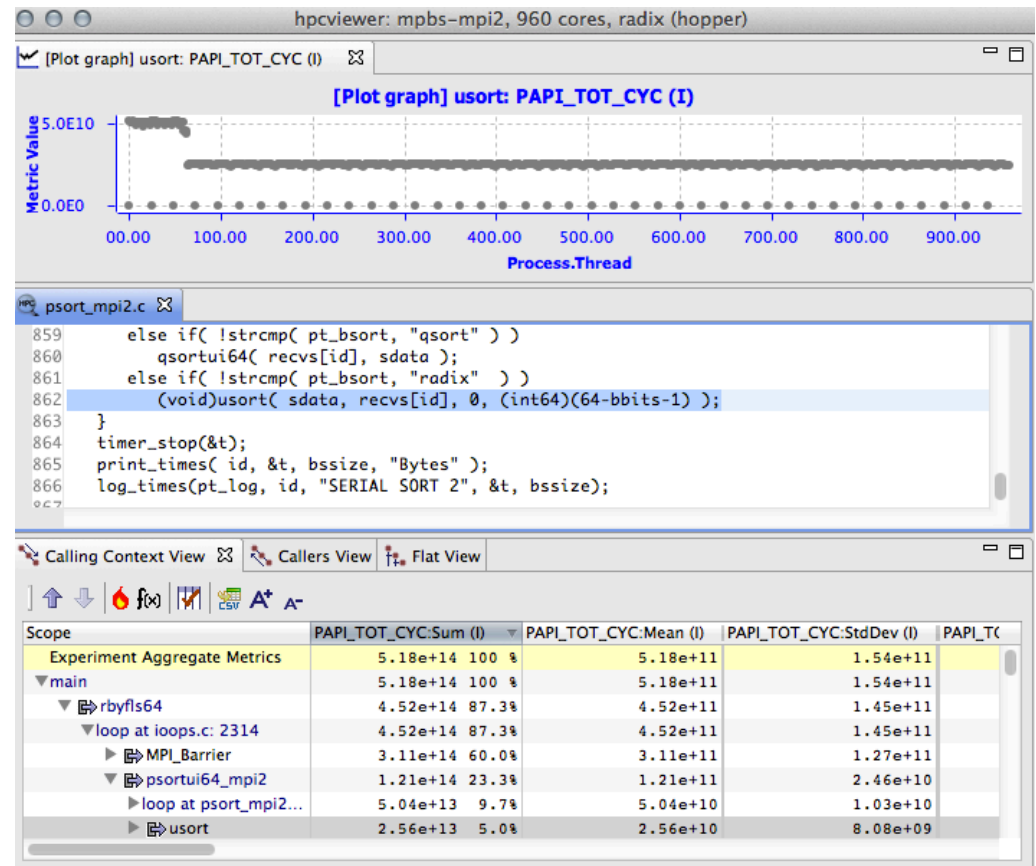
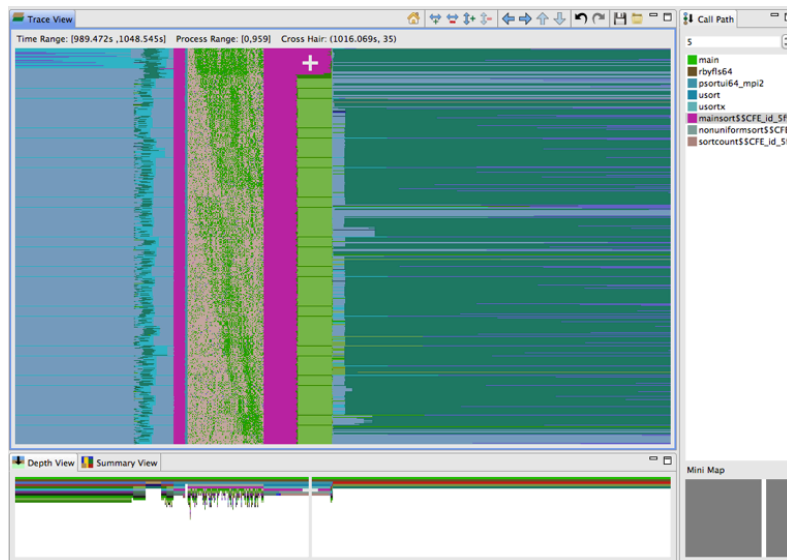


Outline

- Overview of Rice's HPCToolkit
- Pinpointing scalability bottlenecks
 - scalability bottlenecks on large-scale parallel systems
 - scaling on multicore processors
- Understanding temporal behavior
- Assessing process variability
- Understanding threading, GPU, and memory hierarchy
 - blame shifting
 - attributing memory hierarchy costs to data
- Summary and challenges ahead

MPBS @ 960 cores, radix sort

Two views of load imbalance since not on a 2^k cores



Outline

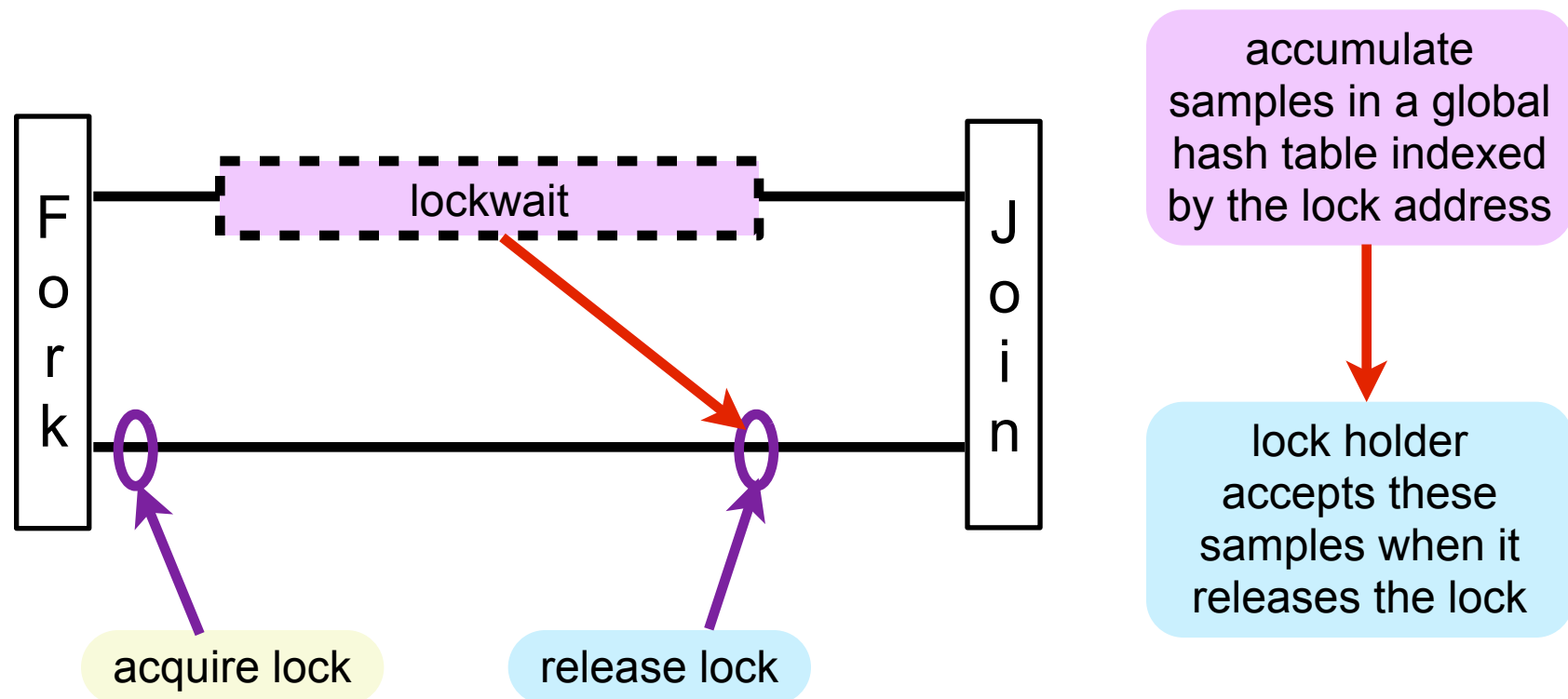
- Overview of Rice's HPCToolkit
- Pinpointing scalability bottlenecks
 - scalability bottlenecks on large-scale parallel systems
 - scaling on multicore processors
- Understanding temporal behavior
- Assessing process variability
- Understanding threading, GPU, and memory hierarchy
 - blame shifting
 - attributing memory hierarchy costs to data
- Summary and challenges ahead

Blame Shifting

- **Problem:** in many circumstances sampling measures symptoms of performance losses rather than causes
 - worker threads waiting for work
 - threads waiting for a lock
 - MPI process waiting for peers in a collective communication
 - idle GPU waiting for work
- **Approach:** shift blame for losses from victims to perpetrators
 - blame code executing while other threads are idle
 - blame code executed by lock holder when thread(s) are waiting
 - blame processes that arrive late to collectives
 - shift blame between CPU and GPU for hybrid code

Directed Blame Shifting

- **Example:**
 - threads waiting at a lock are the symptom
 - the cause is the lock holder
- **Approach: blame lock waiting on lock holder**



Example: Directed Blame Shifting for Locks

Blame a lock holder
for delaying waiting
threads

- Charge all samples that threads receive while awaiting a lock to the lock itself
- When releasing a lock, accept blame at the lock **all of the waiting occurs here (symptom)**

hpcviewer: locktest-2.host

locktest-2.c

```
1#include <omp.h>
2#include "fib.h"
3void g() {
4    int i;
5    omp_lock_t l;
6    omp_init_lock(&l);
7    #pragma omp parallel
8    {
9        #pragma omp master
10       {
11           omp_set_lock(&l);
12           fib(40);
13           omp_unset_lock(&l);
14       }
15       #pragma omp for
16       for(i = 0; i<100; i++) {
17           omp_set_lock(&l);
18           fib(10);
19           omp_unset_lock(&l);
20       }
21   }
22 }
23void f() { g(); }
24int main() { f(); return 0; }
```

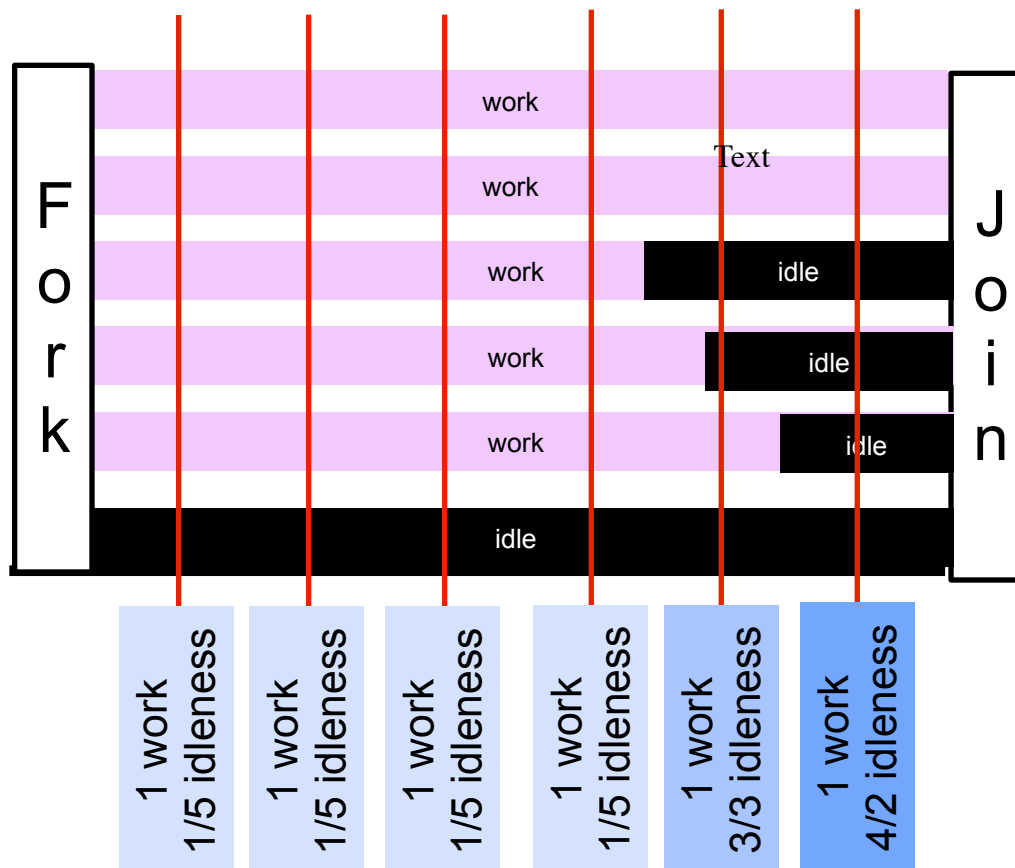
almost all blame for the waiting is attributed here (cause)

Calling Context View

Scope	MUTEX_WAIT:Sum (I)	MUTEX_BLAKE:Sum (I)
Experiment Aggregate Metrics	8.11e+07 100 %	7.93e+07 100 %
monitor_main	8.11e+07 100 %	7.93e+07 100 %
483: main	8.11e+07 100 %	7.93e+07 100 %
29: f	8.11e+07 100 %	7.93e+07 100 %
25: g	8.11e+07 100 %	7.93e+07 100 %
7: L_g_7_par_region0_2_90	8.11e+07 100 %	7.93e+07 100 %
17: kmputc set lock	8.11e+07 100 %	7.93e+07 100 %
12: fib		
20: _kmputc_barrier		
locktest-2.c: 13		7.87e+07 99.2 %

Undirected Blame Shifting

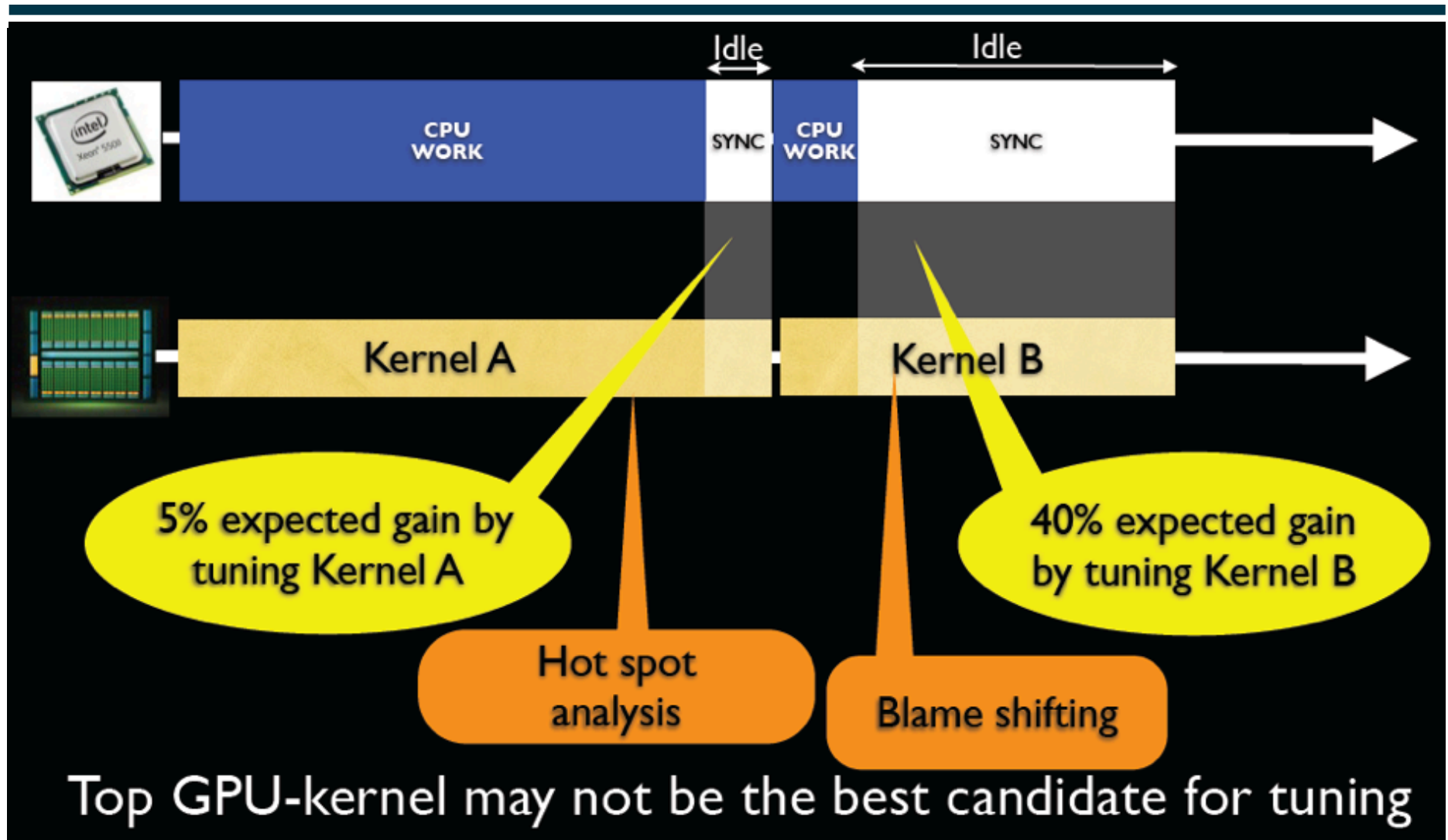
- **Example:**
 - threads idling waiting for work are the symptom
 - the cause is insufficiently parallel work being executed by others
- **Approach:** each working threads proportionally blames itself for instantaneous idling by others when it is sampled



counters hold the number of threads working and idle

working thread charges itself a share of idleness at each sample

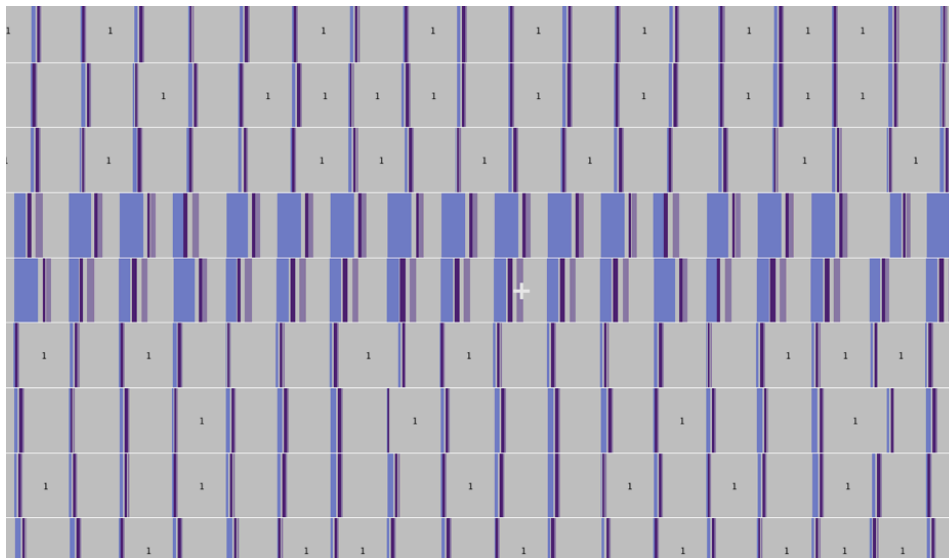
Performance Expectations for Hybrid Code with Blame Shifting



Milind Chabbi, Karthik Murthy, Michael Fagan, and John Mellor-Crummey. Effective Performance Tools for CPU/GPU Systems. SC13. To appear.

GPU Successes with HPCToolkit

- **LAMMPS:** identified hardware problem with Keeneland system
 - improperly seated GPUs were observed to have lower data copy bandwidth



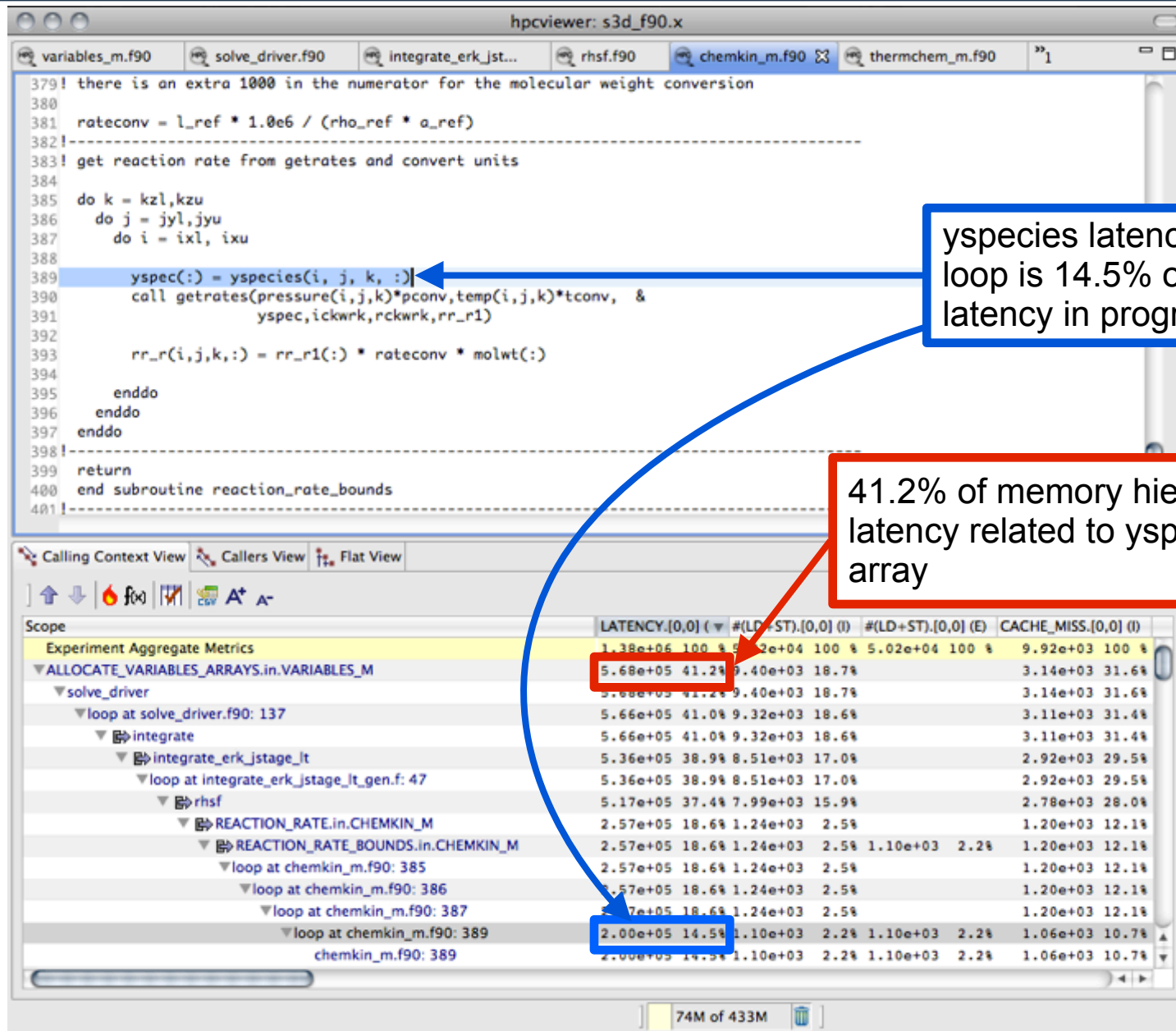
- **LLNL's LULESH:** identified that dynamic memory allocation using `cudaMalloc` and `cudaFree` accounted for 90% of the idleness of the GPU

Data Centric Analysis

- **Goal: associate memory hierarchy performance losses with data**
- **Approach**
 - **intercept allocations to associate with their data ranges**
 - **measure latency with various PMU capabilities**
 - **instruction-based sampling (AMD Opteron)**
 - **precise event-based sampling + load latency facility (Intel)**
 - **marked instructions (IBM Power)**
 - **present quantitative results using hpcviewer**

Xu Liu and John Mellor-Crummey. A Data-centric Profiler for Parallel Programs. SC13. To appear.

Data Centric Analysis of S3D



yspecies latency for this loop is 14.5% of total latency in program

41.2% of memory hierarchy latency related to yspecies array

Outline

- Overview of Rice's HPCToolkit
- Pinpointing scalability bottlenecks
 - scalability bottlenecks on large-scale parallel systems
 - scaling on multicore processors
- Understanding temporal behavior
- Assessing process variability
- Understanding threading, GPU, and memory hierarchy
 - blame shifting
 - attributing memory hierarchy costs to data
- Summary and challenges ahead

Summary

- **Sampling provides low overhead measurement**
- **Call path profiling + binary analysis + blame shifting = insight**
 - scalability bottlenecks
 - where insufficient parallelism lurks
 - sources of lock contention
 - load imbalance
 - temporal dynamics
 - bottlenecks in hybrid code
 - problematic data structures
 - hardware counters for detailed diagnosis
- **Other capabilities**
 - attribute memory leaks back to their full calling context

HPCToolkit Status

- **Operational today on**
 - 64- and 32-bit x86 systems running Linux (including Cray XT/E/K)
 - IBM Blue Gene
 - IBM Power7 systems running Linux
- **Available as open source software at <http://hpctoolkit.org>**
- **Emerging capabilities**
 - **NVIDIA GPU**
 - measurement and reporting using GPU hardware counters
 - data centric analysis
 - OpenMP analysis using OMPT

OMPT: Emerging Monitoring for OpenMP

2-hpcviewer: LULESH_OMP.host

LULESH_OMP.cpp

```

1287 /*****
1288  /*    compute the hourglass modes */
1289
1290
1291 #pragma omp parallel for firstprivate(numElem, hourg)
1292     for(Index_t i2=0; i2<numElem; ++i2){
1293         Real_t *fx_local, *fy_local, *fz_local ;
1294         Real_t hgfx[8], hgy[8], hgfh[8] ;
1295
1296         Real_t coefficient;
1297
1298         Real_t hourg0[547], hourg1[547], hourg2[547], hourg3[547],

```

Problem: calling context for parallel regions and tasks is not readily available to tools

Calling Context View Callers View Flat View

Scope

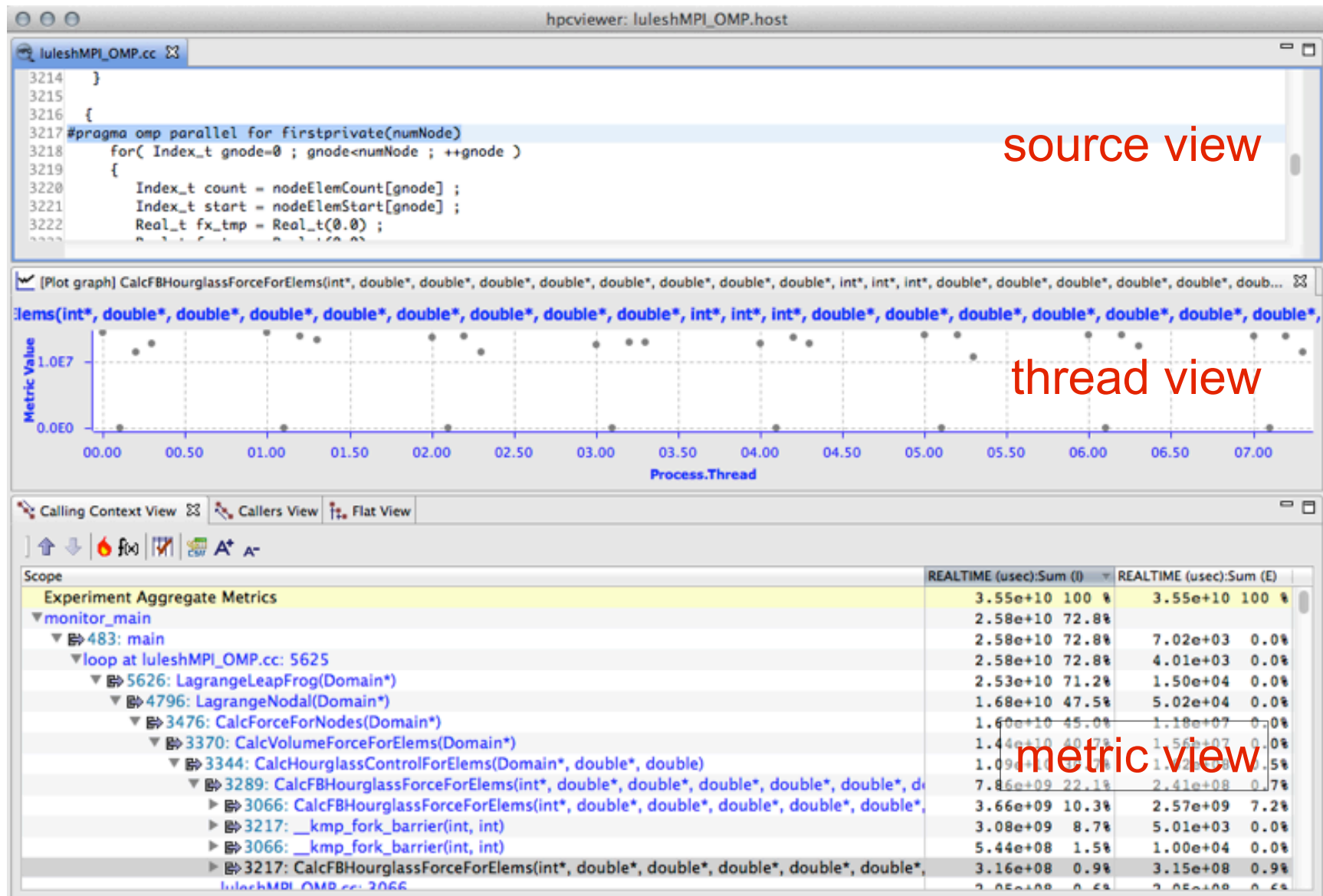
Scope	REALTIME (usec):Sum (I)	REALTIME (usec):Sum (E)
Experiment Aggregate Metrics	6.32e+08 100 %	6.32e+08 100 %
▼ monitor_begin_thread	6.06e+08 95.8%	
▼ 940: __kmp_launch_worker(void*)	5.80e+08 91.8%	
▼ 729: __kmp_launch_thread	5.80e+08 91.8%	1.51e+04 0.0%
▼ 6314: __kmp_invoke_task_func	3.38e+08 53.5%	
▼ 7586: L kmp invoke pass parms	3.38e+08 53.5%	
▶ L_Z28CalcFBHourglassForceForElemsPdS_S_S_S_d_1291__par_loop0_2_276	6.48e+07 10.3%	4.14e+07 6.5%
▶ L_Z22CalcKinematicsForElemsid_1931__par_loop0_2_855	5.36e+07 8.5%	1.72e+07 2.7%
▶ L_Z28CalcHourglassControlForElemsPdd_1516__par_loop0_2_424	4.73e+07 7.5%	1.64e+07 2.6%
▶ L_Z23IntegrateStressForElemsPdS_S_S_864__par_loop0_2_125	4.34e+07 6.9%	8.66e+06 1.4%
▶ L_Z31CalcMonotonicQGradientsForElemsv_2040__par_loop0_2_965	2.82e+07 4.5%	1.59e+07 2.5%
...		
▶ 6333: __kmp_join_barrier(int)	1.63e+07 2.6%	2.50e+04 0.0%
▶ 6302: __kmp_clear_x87_fpu_status_word	2.00e+04 0.0%	2.00e+04 0.0%
kmp_runtime.c: 6236		
▶ 940: __kmp_launch_monitor(void*)	2.53e+07 4.0%	
▼ monitor_main	2.63e+07 4.2%	
▼ 483: main	2.63e+07 4.2%	2.10e+05 0.0%
▶ 3187: LagrangeLeapFrog()	2.52e+07 4.0%	
▶ 3049: Domain::AllocateNodeElemIndexes()	4.66e+05 0.1%	2.15e+05 0.0%
▶ 2995: Domain::AllocateElemPersistent(unsigned long)	8.09e+04 0.0%	

Key OMPT Design Objectives

- **Enable tools to gather information and associate costs with application source and runtime system**
 - **provide interface for low-overhead sampling-based tools**
 - **enable tools to reconstruct application-level profiles**
 - **alternative to implementation-level view**
 - **associate activity of a thread at any point in time with a state**
 - **enable performance tools to monitor behavior**
- **Negligible overhead if OMPT interface is not in use**
- **Define support for trace-based performance tools**

Integrated View of MPI+OpenMP with OMPT

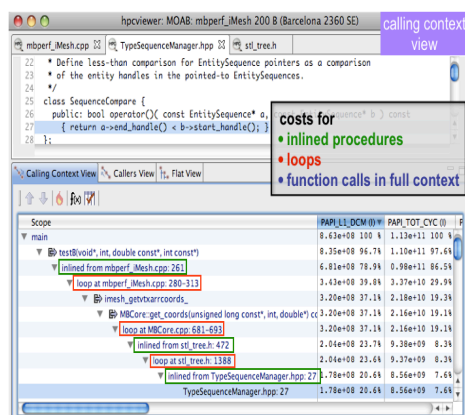
LLNL's IuleshMPI_OMP (8 MPI x 3 OMP), 30, REALTIME@1000



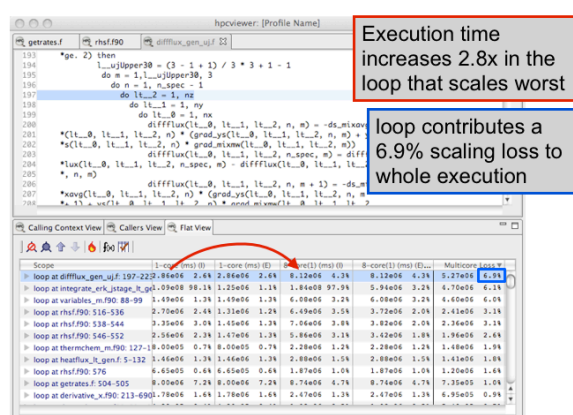
Tool Challenges Ahead

- **Address challenges of emerging systems**
 - heterogeneity (e.g., on-chip; host + accelerator)
 - growth in thread counts: MIC supports 200+ threads
 - increasing scale of systems (e.g., Sequoia)
- **Identify causes rather than symptoms (blame shifting)**
- **Measure and analyze all facets of application performance**
 - CPU, accelerator, data movement, synchronization, I/O, power
 - interactions: HW, other jobs, system software
- **Analyze asynchronous activities**
- **Support dynamic adaptation of software**
 - measurements and decision algorithms to drive adaptation
 - assessment of adaptation policies
- **Provide analysis to support higher level insight, diagnosis, and guidance**

HPCToolkit Capabilities at a Glance



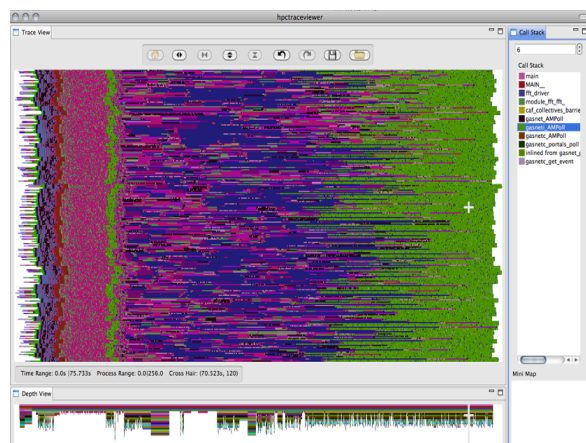
Attribute Costs to Code



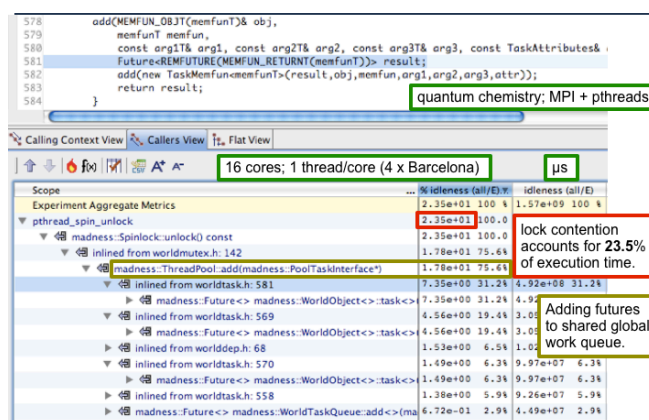
Pinpoint & Quantify Scaling Bottlenecks



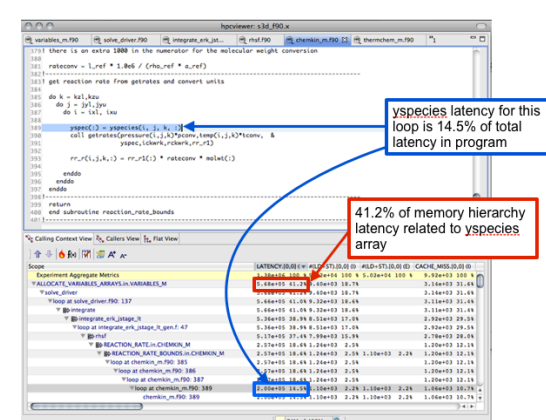
Assess Imbalance and Variability



Analyze Behavior over Time



Shift Blame from Symptoms to Causes



Associate Costs with Data

hpctoolkit.org